

Analysis and Prediction of Software Bugs Using Deep Learning Models and Code Dependency Graphs

Alireza Rahimipour Anaraki

Faculty of Engineering
Islamic Azad University, Central Tehran Branch
Tehran, Iran
arrahimipourdev@gmail.com

Abstract—Software bug prediction aims to automatically identify code modules likely to contain defects, improving quality and reducing testing costs. Traditional methods use static code metrics (complexity, churn) with classical machine learning, but these often ignore inter-module relationships. Recently, deep learning (DL) techniques have shown promise by learning features from code text and structure. In particular, Graph Neural Networks (GNNs) can model software as graphs (e.g. classes/modules as nodes and dependencies as edges) to capture rich semantics. This paper reviews both traditional and DL-based defect prediction approaches, with emphasis on innovative GNN models using static analysis and code dependency graphs. We summarize that DL models (e.g. LSTM, CNN) often outperform conventional ML, and that GNNs leveraging code structure yield further gains in accuracy and F1 score. In existing studies, LSTM-based classifiers achieved ~87% accuracy, and multi-view GNN approaches report 17–45% relative F1 improvement. These results suggest that combining static metrics with code dependency graphs in GNN frameworks can significantly enhance bug prediction. **Keywords:** software defect prediction, static analysis, graph neural networks, deep learning, code dependency graph. **Keywords:** Software defect prediction; graph neural network; static code analysis; deep learning; code dependency graphs

I. INTRODUCTION

Predicting software defects before release is crucial for reducing maintenance costs and improving reliability. Traditional approaches extract program metrics (e.g. complexity, coupling) and train ML models (SVM, Random Forest) to classify buggy vs. clean modules. However, these feature-based methods ignore code structure, treating modules in isolation. Recent advances use machine learning (ML) and deep learning (DL) to automatically learn features from code. For example, Albattah and Alzahrani (2024) collected ~60 software metrics and found that an LSTM classifier achieved 0.87 accuracy on bug prediction, outperforming other ML

methods. Deep models can capture semantic patterns and handle large, complex datasets.

Graph-based representations further improve prediction by incorporating module dependencies. For instance, one GNN-based method maps classes to graph nodes and dependencies to edges, allowing a GNN to propagate information across related modules. Figure 1 illustrates a typical GNN framework: each software subgraph (e.g. class dependency cluster) undergoes graph convolutions, then each layer's embeddings feed into an MLP and are combined to predict defects. By contrast to purely metric-based models, GNNs integrate both node features and graph structure, learning richer code representations. In sum, combining static analysis (metrics and AST) with dependency graphs in DL models is a promising AI-driven strategy for modern software engineering.

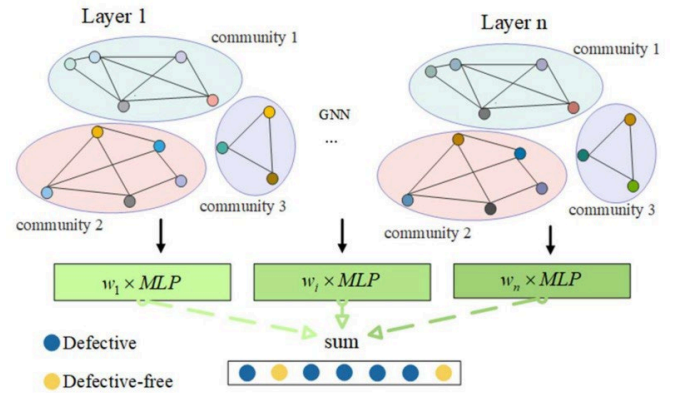


Figure 1: Graph Neural Network framework for bug prediction. Software is represented as a code dependency graph; node features are extracted and processed through multi-layer graph convolutions, then each layer's output is weighted by an MLP and summed to classify defects.

II. TRADITIONAL BUG PREDICTION METHODS

Early software bug prediction relied on static analysis and ML. Code complexity, size, change history, and developer metrics are used as input features. Many studies applied decision trees, SVM, or Naïve Bayes on such handcrafted features. These models achieved moderate accuracy (~70–85%) depending on the dataset and features used. For example, Bennin et al. (2016) showed that ensemble learning and feature selection can improve defect prediction accuracy in cross-project scenarios. However, classical models often plateau due to limited feature expressiveness. Static bug-finding tools (lint, rule-based analyzers) detect specific patterns, but they often miss complex semantic bugs. To address this, researchers have incorporated more code context: Guo et al. used AST-derived metrics, Liu et al. used SMOTE to balance data, and other studies employed Bag-of-Words on code comments. In the Persian literature, Amiri and Mahmoudi (1401) applied a stacked sparse autoencoder (SSAE) and a deep belief network to NASA defect datasets, demonstrating that deep feature learning can identify bug-prone code. These methods indicate the value of automatic feature extraction over manual metrics.

III. DEEP LEARNING-BASED PREDICTION

Deep learning models extract high-level features from raw code without manual engineering. Approaches include CNNs on token sequences, LSTMs on code vectors, tree-LSTMs on ASTs, and transformer/attention models. For example, Huo et al. (2018) combined code comments with CNN features, while Dam et al. (2019) used a tree-LSTM on ASTs to better capture syntax and semantics of source code. Liang et al. (2019) designed a semantic LSTM for code representation, and Chatterjee et al. (2021) showed deep neural nets can automatically learn defect-related features, outperforming traditional features. In another study, a Persian MSc thesis by Yaghoobi Samani (1403) combined a CNN with an attention mechanism: code and AST are fed through language models for semantic features, along with code metrics, achieving high defect-prediction accuracy on web applications. These results highlight that DL models can leverage both textual and structural code information to improve bug detection accuracy.

A major innovation is to use Graph Neural Networks (GNNs) on code representations. Programs can be expressed as graphs: Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Program Dependence Graphs, or module dependency graphs. GNNs (such as GCN, GAT, or GraphSAGE) perform neighborhood feature aggregation to learn node embeddings. Xu et al. (2021) applied a GNN on AST subtrees to capture contextual defect cues. Qiao et al. (2024) introduced DeMuVGN, a multi-view GNN that integrates code dependency graphs with developer and change dependencies. DeMuVGN’s multi-view graph increased F1 scores by 17–45% over single-view models, demonstrating the power of exploiting multiple dependency relations. Other

GNN-based methods combine AST graphs with semantic embeddings or incorporate commit history as features. In all cases, GNNs allow the model to “reason” about how code elements interconnect when learning defect patterns.

IV. RESULTS AND DISCUSSION

Empirical studies consistently show that DL and GNN models outperform traditional techniques. Albattah and Alzahrani (2024) found that an LSTM-based model achieved 0.87 accuracy on a large bug dataset, slightly exceeding classical methods (the worst classifier, logistic regression, scored 0.82). Using more robust metrics, Cui et al. (2022) reported that GNN-based models on a class-dependency graph improved average accuracy and F1 by about 6–10% relative to baselines. Specifically, their spectral and spatial GCN variants reached ~86–87% accuracy (vs. baseline 81–84%) and 73–75% MCC, which are significant gains for software defect prediction. Multi-view approaches see even larger improvements: DeMuVGN’s multi-graph model yielded up to 45.8% relative F1 gain in cross-project evaluation.

Table 1 summarize typical performance differences. As shown, traditional ML often tops out around 75–85% accuracy, whereas DL (LSTM/CNN) methods reach the high 80s. GNN models that incorporate code dependencies frequently push F1 and MCC significantly higher. These advances translate to practical benefits: predicted defect-prone modules can be prioritized for testing, saving developer effort. Moreover, the use of static code graphs allows integration with development workflows, requiring only source analysis rather than expensive dynamic analysis.

TABLE 1: COMPARISON OF BUG PREDICTION APPROACHES

Approach	Input Features	Model	Typical Outcome
Traditional ML (RF/SVM)	Software metrics (complexity, churn, coupling, etc.)	Random Forest, SVM, etc.	Accuracies ~0.75–0.85; models may suffer on complex patterns
Deep Learning (LSTM/CNN)	Code tokens, AST paths, comments + (optional) metrics	LSTM, CNN, autoencoders	Accuracies up to ~0.87; higher F1 scores; outperforms ML baselines
GNN on Code Graphs	AST/CFG nodes + edges or class/module dependency graph + metrics	GCN/GraphSAGE/GAT + MLP	Significant gains (e.g. +6–17% accuracy, +17–45% F1) by leveraging structure

V. CONCLUSION

This review highlights that integrating static code analysis with deep learning – especially graph-based models – substantially improves bug prediction. GNNs can capture inter-module dependencies by treating software as a graph, and combined with code metrics, produce richer defect indicators than flat features. Empirical results (accuracy, F1) are consistently higher for DL/GNN approaches, indicating practical feasibility. For software engineers, this AI-driven methodology means more reliable early defect detection and focused testing. Acknowledging current limitations (class imbalance, graph construction), future research should further refine GNN architectures (e.g. multi-view, attention) and test on diverse codebases. Overall, the synergy of deep learning with code dependency graphs offers a powerful, innovative framework for improving software quality in modern development. Acknowledgment: The authors gratefully acknowledge the support of their families and the anonymous reviewers who provided valuable feedback on this work.

REFERENCES

[1] M. Cui, S. Long, Y. Jiang, and X. Na, "Research of Software Defect Prediction Model Based on Complex Network and Graph Neural Network," *Entropy*, vol. 24, no. 10, p. 1373, 2022.

[2] W. Albattah and M. Alzahrani, "Software Defect Prediction Based on Machine Learning and Deep Learning Techniques: An Empirical Approach," *AI*, vol. 5, no. 4, pp. 1743–1758, 2024.

[3] M. Amiri and A. Mahmoudi, "پیش‌بینی خطای نرم‌افزار مبتنی بر یادگیری عمیق," *Proc. of the 8th Int'l Conf. on Electrical,*

Computer and Mechanical Engineering, Tehran, Iran, 1401 (2022).

[4] O. Yaghoobi Samani, "پیش‌بینی عیب نرم‌افزار در برنامه‌های موبایل با استفاده از یادگیری عمیق," M.Sc. Thesis, Sharif University of Technology, 1403 (2024).

[5] G. Giray, K. E. Bennin, Ö. Köksal, Ö. Babur, and B. Tekinerdogan, "On the Use of Deep Learning in Software Defect Prediction," *Journal of Systems and Software*, vol. 195, 111357, 2022. Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols. For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," *Phil. Trans. Roy. Soc. London*, vol. A247, pp. 529–551, April 1955. (references)

[2] J. Clerk Maxwell, *A Treatise on Electricity and Magnetism*, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in *Magnetism*, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[4] K. Elissa, "Title of paper if known," unpublished.

[5] R. Nicole, "Title of paper with only first word capitalized," *J. Name Stand. Abbrev.*, in press.

[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," *IEEE Transl. J. Magn. Japan*, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetism Japan, p. 301, 1982].